**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное автономное образовательное учреждение**

**высшего образования**

**«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И

ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАФЕДРА АНАЛИЗА ДАННЫХ И

ТЕХНОЛОГИЙ ПРОГРАММИРОВАНИЯ

Направление: 09.03.03 – Прикладная информатика

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

**Разработка приложения для проектирования ИИ моделей для участия в гоночной игре Trackmania с использованием машинного обучения**

**Работа завершена:**

Студент 4 курса

группы 09-052

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г.    \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Близнюк В.Ю.

**Работа допущена к защите:**

Научный руководитель

к.т.н., доцент

кафедры анализа данных и

технологий программирования

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г.    \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Сабитов Ш.Р.

Заведующий кафедрой

к.ф.-м.н.

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г.    \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Бандеров В.В.

Казань-2024

Содержание

[Аннотация 3](#_Toc168417773)

[Введение 5](#_Toc168417774)

[1. Исследование предметной области создания искусственного интеллекта для гоночных игр 7](#_Toc168417775)

[1.1. Анализ предметной области 7](#_Toc168417776)

[1.2. Обзор искусственного интеллекта в большинстве гоночных видеоигр на примере серии видеоигр Need For Speed 8](#_Toc168417777)

[1.3. Обзор драйватаров в серии видеоигр Forza 9](#_Toc168417778)

[1.4. Обзор механики призраков в видеоигре TrackMania 11](#_Toc168417779)

[1.5. Обзор TMRL 12](#_Toc168417780)

[1.6. Описание программных средств реализации 12](#_Toc168417781)

[2. Проектирование приложения для создания моделей 14](#_Toc168417782)

[2.1. Составление технического задания 14](#_Toc168417783)

[3. Разработка приложения для обучения и исполнения моделей 15](#_Toc168417784)

[3.1. Реализация связи приложения и видеоигры 15](#_Toc168417785)

[3.2. Создание gym окружения видеоигры для обучения 19](#_Toc168417786)

[3.3. Реализация нейронной сети 21](#_Toc168417787)

[3.4. Заключение по приложению и циклу его работы 30](#_Toc168417788)

[4. Обучение и тестирование моделей 33](#_Toc168417789)

[4.1. Обучение на прямой карте 33](#_Toc168417790)

[5. Заключение 40](#_Toc168417791)

[Список использованных источников 42](#_Toc168417792)

# Аннотация

Выпускная квалификационная работа по теме «Разработка приложения для проектирования ИИ моделей для участия в гоночной игре Trackmania с использованием машинного обучения» состоит из 43-ех страниц текста, 22-ух рисунков и 13-и использованных источников. Структура работы включает введение, 4 главы, заключение и список использованных источников.

Целью работы является создание приложения для проектирования ИИ моделей для участия в гоночной игре Trackmania с использованием машинного обучения.

Первая глава содержит анализ предметной области, обзор аналогов и описание программных средств реализации. Вторая глава включает в себя составление технического задания. В третьей главе описаны процессы разработки приложения и всех его компонентов: алгоритма обучения, окружения для него и связи между приложением и видеоигрой. Четвертая глава посвящена процессу обучения моделей, а также тестированию приложения и моделей. В заключении перечислены основные результаты выполнения выпускной квалификационной работы.

Ключевые слова: машинное обучение с подкреплением, DQN, TrackMania Nations Forever, TMInterface, Python, PyTorch, gymnasium.

Abstract

The final qualification work on the topic «Development of an application for designing AI models to participate in the racing game Trackmania using machine learning» consists of 43 pages of text, 22 drawings and 13 used sources. The structure of the work includes an introduction, 4 chapters, a conclusion and a list of references.

The aim of the work is to develop an application for designing AI models to participate in the racing game Trackmania using machine learning.

The first chapter contains an analysis of the subject area, an overview of analogs and a description of software implementation tools. The second chapter includes the preparation of the terms of reference. The third chapter describes the processes of developing the application and all of its components: the learning algorithm, the environment for it, and the communication between the application and the video game. Fourth chapter focuses on the model training process and testing of the application and models. In the conclusion, the main results of the final qualifying work are listed.

Key words: Depp Reinforcement Learning, DQN, TrackMania Nations Forever, TMInterface, Python, PyTorch, gymnasium.

# Введение

Область разработки искусственного интеллекта и машинного обучения активно развивается. С каждым днём появляются новости о новых языковых моделях, улучшенных алгоритмах обучения роботов или дронов. В условиях стремительного развития технологий и увеличения интереса к искусственному интеллекту, создание систем, способных эффективно взаимодействовать с игровыми средами, становится важной задачей. На данный момент внутриигровые соперники ограничиваются либо ботами со строго прописанными правилами, либо рекордами игроков. Оба подхода имеют свои минусы. Неживые соперники не могут адаптироваться к новым условиям и не предоставляют ощущения соревновательности, которое предлагает гонка с живыми игроками. Записанные рекорды дают это чувство, но сражение происходит не с человеком, а с его «призраком», который только повторяет действия игрока. И это не говоря про возможность столкновения с машиной соперника, что не представляется возможным из-за нематериальности «призрака».

На сегодняшний день существует небольшое количество исследований и приложений, посвященных разработке искусственного интеллекта для гоночных игр, которые могли бы решить приведенные проблемы. Например, Playground Games с их серией видеоигр Forza, используют механику драйватаров, в которой анализируется поведение игроков, после чего неживые противники пытаются имитировать то, как водит соответствующий боту человек.

Однако искусственный интеллект, основанный на нейронных сетях, чаще всего встречается в пользовательских проектах. Однако и они обладают своими недостатками, основной из которых – отсутствие доступа к состоянию игры, из-за чего входные данные нейронной сети приходится получать из частых снимков экрана запущенной игры, а сама видеоигра получает входные данные с помощью имитации использования клавиатуры или геймпада.

Актуальность данной работы заключается в том, что на сегодняшний день не существует приложений, которые могли бы имитировать поведение реального игрока в гоночные видеоигры, то есть искусственный интеллект которых был бы основан на использовании нейронных сетей, при этом имея доступ к состоянию видеоигры и возможность отдавать приказы напрямую, а не через эмуляцию устройства ввода. В процессе консультации с научным руководителем была поставлена цель и выделены задачи. Была налажена регулярная коммуникация с научным руководителем для уточнения вопросов и консультаций.

Целью ВКР является создание приложение для создания моделей искусственного интеллекта с помощью машинного обучения, которые могут быть использованы в гоночной игре TrackMania Nations Forever.

Задачи выпускной квалификационной работы:

1. Провести анализ предметной области;
2. Провести обзор существующих аналогов;
3. Составить техническое задание к функциональной части;
4. Описать программные средства реализации приложения;
5. Приготовить видеоигру и средства ввода и получения данных;
6. Разработать нейронную сеть и скрипт обучения;
7. Протестировать полученные модели против реальных игроков.

Выпускная квалификационная работа состоит из 4-ех глав, введения, заключения, списка использованной источников и приложений. В работе описан весь процесс проектирования, конструирования и разработки приложения, представлены скриншоты результатов обучения моделей нейронной сети, описания классов, а также вставлены наиболее важные части кода приложения.

# 1. Исследование предметной области создания искусственного интеллекта для гоночных игр

# 1.1. Анализ предметной области

При анализе существующих решений были выделены закономерности подходов к созданию искусственного интеллекта в гоночных видеоиграх. Большинство игр имеют интеллект соперников, напрямую прописанный в коде, никак не обучающийся и не адаптирующийся, то есть не использующий нейронные сети. Использование подобных средств редко и часто не реализует весь потенциал машинного обучения. Помимо этого, четко фиксированные правила сложны для разработки, так как, в зависимости от сложности физической модели видеоигры, нужно учитывать множество различных параметров: от положения и поворота машины на трассе до состояния дорожного покрытия и погоды.

Если же разработчики видеоигры используют машинное обучение, то часто оно ограничено или до подсказок самому игроку в том, как лучше проехать тот или иной поворот, или до незаметной помощи ботам. Во втором случае, однако, это часто имеет вспомогательный и подсказывающий характер, который с некоторым шансом влияет на решение, принятое игрой на основе предписанных правил.

Пользовательские и фанатские проекты же, которые используют нейронные сети, имеют проблему в виде невозможности напрямую получать состояние видеоигры или отдавать ей команды для движения автомобиля. Чтобы иметь входные данные для обучения и использования модели, приходится делать десятки снимков экрана в секунду и пытаться их обработать, чтобы модель получала только важные и необходимые данные. Очевидно, это одновременно ресурсозатратно и сложно для реализации. Данные, полученные со скриншота, могут быть неполными или обработанными неправильно, что также могут повлиять на работу искусственного интеллекта. Второй проблемой является ввод действий в игру, что часто приходится имитировать с помощью эмуляции устройства ввода и его нажатий.

Некоторые видеоигры, которые предполагают соревнование в том, кто проедет трассу быстрее, а не первее, такие как TrackMania, часто имеют механику призраков – предзаписанных другим игроком действий того, как он сам проходил трек. Их проблема заключается в том, что такие соперники должны создаваться отдельно для каждой карты, машины или другого фактора. Изменение даже одного из них может повлиять на поведение призрака и сделать его несостоятельным. Кроме того, это не позволяет выбрать сложность соперника, если трасса новая или непопулярная, и не дает возможности сталкиваться с машиной другого гонщика.

# 1.2. Обзор искусственного интеллекта в большинстве гоночных видеоигр на примере серии видеоигр Need For Speed

Искусственный интеллект большинства видеоигр создан с помощью большого и четкого набора правил, которых придерживаются внутриигровые соперники. Так было с создания самых первых игр этого жанра, так как на тот момент использование нейронных сетей было мало распространено и не было эффективным с производительность персональных компьютеров и консолей тех лет.

Из-за этого для каждой ситуации на дороге в коде видеоигр прописано то, как должен действовать гонщик. Положение каждой машины на трассе, их скорости и модели, текущая погода, всё это влияет на поведение машины и прописано сложной системой условных выражений. Конечно, создавать такую систему очень сложно, так как нужно учитывать все вышеперечисленные факторы, влияющие на то, как будет реагировать на ввод машина.

Так, на рисунке 1.1 машины на 192 и 193 местах выбирают то, как нужно действовать, не на основе своих знаний или благодаря данным от обучения, а от предписанных в коде правил.



Рисунок 1.1. Скриншот из видеоигры Need For Speed: The Run

# 1.3. Обзор драйватаров в серии видеоигр Forza

PlayGround Games с их серией видеоигр Forza решили шагнуть дальше и уже в первой своей видеоигре Forza Motorsport, выпущенной в 2005 году, ввели механику драйватаров. Во время прохождения трассы игра изучала поведение игрока и использовала полученные данные, чтобы создать внутриигрового соперника со стилем вождения, как у реального человека.

Тогда драйватары сохранялись локально, и чтобы сразиться со своим другом, приходилось копировать файлы сохранения. Это изменилось с выходом Forza Motorsport 5 в 2013 году, когда данные о драйватарах сохранялись в облаке. Это позволило участвовать в гонках сразу с несколькими соперниками, которые хоть и управлялись компьютером, но пытались имитировать поведение соответствующих себе живых игроков.

Однако система не является совершенной. Драйватары не используются для непосредственного принятия решения на трассе, а лишь дают подсказки обыкновенному искусственному интеллекту. Несовершенство такого решения также подтверждается тем, что даже в последних играх серии сложность влияет не только на то, какие драйватары будут использованы в гонке, но и на их автомобили, искусственно замедляя или ускоряя соперников в зависимости от сложности и игровой ситуации.

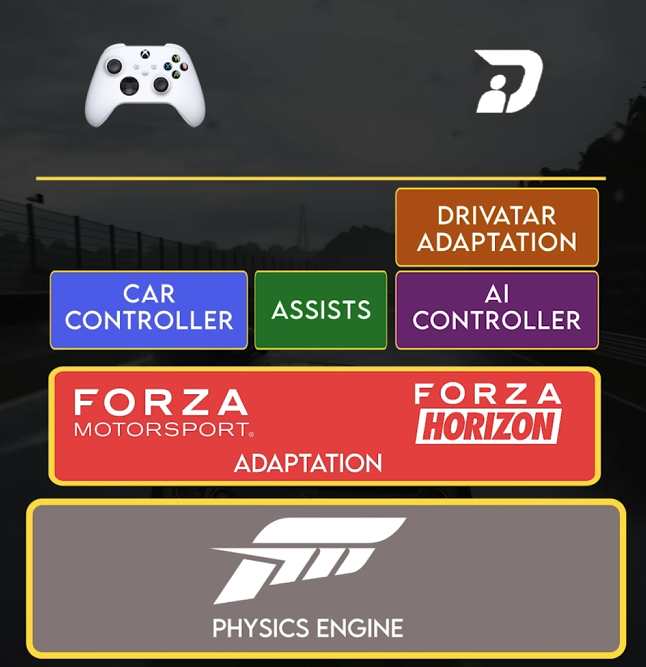


Рисунок 1.2. Система управления автомобилем в играх серии Forza

# 1.4. Обзор механики призраков в видеоигре TrackMania

Nadeo в своей видеоигре TrackMania совсем отказались от физических соперников на трассе, создав игру для того, чтобы игроки соревновались в рекордах по времени прохождения, а не в том, кто кого обогнал.

Такие рекорды позволяют получить чувство соревнования с другими живыми игроками, однако они не появляются на трассе в виде физической машины с коллизиями, что портит ощущение погружения в игровой процесс. Другой проблемой является то, что рекорды должны быть записаны конкретными людьми на конкретных трассах в конкретных условиях. Очевидно, что в такой ситуации выбор сложности для непопулярных и новых карт не представляется возможным.



Рисунок 1.3. Демонстрация призраков в гонке TrackMania Nations Forever

# 1.5. Обзор TMRL

TMRL – пользовательский фреймворк для создания приложений для обучения с подкреплением, которое идет со встроенными возможностями подключения фреймворка к видеоигре TrackMania 2020.

На странице проекта демонстрируются возможности фреймворка на примере создания рекордов, способных проходить трассы различной сложности и правильно оценивающих окружающую обстановку.

Однако, данные фреймворк получает с помощью последовательных снимков экрана и их последующей обработки, а ввод в игру имитируется эмуляцией геймпада, из-за чего программу во время обучения или выполнения нельзя сворачивать, а скриншоты и их обработка могут давать ложные результаты.

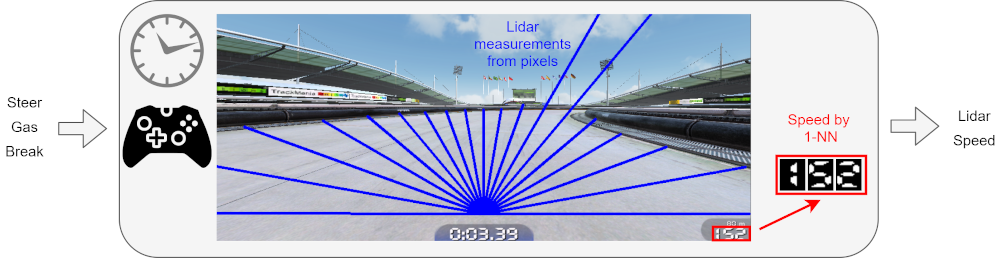


Рисунок 1.4. Демонстрация получения и ввода данных в TMRL

# 1.6. Описание программных средств реализации

В качестве видеоигры, для которой приложение будет создавать модели, была выбрана TrackMania Nations Forever, так как игра имеет детерминистскую машину состояний и четкую физическую модель, что позволяет модели нейронной сети обучаться без опасения натолкнуться на какое-то случайное событие. Для получения данных игры и ввода в неё было решено использовать TMInterface. Для работы с ней будет использована соответствующая Python библиотека.

В качестве языка разработки был выбран Python, а библиотеки для создания нейронной сети – Torch. Для создания окружения для нейронной сети использована библиотека gym.

# 2. Проектирование приложения для создания моделей

# 2.1. Составление технического задания

В результате анализа предметной области, цели и поставленных задач возникла необходимость разработки приложения, позволяющего создавать и сохранять модели нейронной сети, способной проходить трассы в гоночной видеоигре TrackMania Nations Forever. Всё приложение должно быть написано на языке программирования Python. Связь приложения с видеоигрой должна осуществляться с помощью TMInterface, окружение для нейронной сети должно быть создано с помощью библиотеки gym, нейронная сеть должна быть написана с помощью библиотеки Torch.

Приложение должно представлять собой программу, которая при введенных данных о треке и, при необходимости, сохраненных данных обучения, должна начать процесс обучения нейронной сети для участия в гонке и её прохождения. При наличии сохраненных данных обучения программа также должна иметь возможность загрузить модель и использовать ее для прохождения трассы.

Обучение должно выполняться автоматически и без необходимости участия человека или открытого окна видеоигры или программы. Созданные модели должны уметь проходить трассы, прохождение которых человеком занимает 15 секунд, за время, не превышающее 30 секунд. Обучение модели, достигающей такого результата, должно занимать не более 8 часов. Приложение должно давать возможность получить данные о треке при указании пути до файла, которых должно быть достаточно для того, чтобы ввести их при начале обучения или исполнения модели.

# 3. Разработка приложения для обучения и исполнения моделей

# 3.1. Реализация связи приложения и видеоигры

Исходя из технического задания, были написаны необходимые классы и интерфейсы, которые обеспечивали доступ из приложения к состоянию видеоигры и возможность ввода для управления машиной.

Использовалась пользовательская модификация видеоигры TrackMania Nations Forever под названием TMInterface, которая позволяет наладить связь с собой с помощью одноименной библиотеки для Python.

Основными классами библиотеки, которые использовались в приложении, являются:

* + Client – класс, напрямую связанный с клиентом игры. Предназначен в первую очередь для наследования от себя и переопределения методов обратного вызова.
  + TMInterface – основной класс, который позволяет получать состояние игры или менять его, а также взаимодействовать с самими процессами видеоигры.
  + SimStateData – класс, который хранит в себе состояние видеоигры.
  + HmsDynaStruct – класс, который хранит в себе состояние об автомобиле в конкретный момент времени.
  + CheckpointData – класс, который хранит в себе состояние о пройденных чекпоинтах.

Для наследования от класса Client был создан класс SimStateClient. Он хранит в себе поля, отвечающие за хранение ссылки на объекты классов TMInterface, SimStateData и CheckpointData, данные, которые должны быть введены в игру, факт завершения гонки и флаг, при поднятии которого клиент должен перезапустить гонку.

Было переопределено 2 метода класса Client:

* + on\_run\_step – вызывается каждый физический тик в гонке, имеет в параметрах TMInterface, который хранит все текущие состояния. Метод используется для сохранения в клиенте ссылок на TMInterface, SimStateData и CheckpointData. Также при наличии ввода передает его в игру, а при необходимости перезапуска гонки – перезапускает её.
  + on\_checkpoint\_count\_changed – вызывается при прохождении чекпоинта в гонке, в том числе финишной черты. Используется для того, чтобы гонка не завершалась при проезде финиша, при этом меняя поле класса, отвечающее за факт конца гонки.

Также был создан класс SimStateInterface, обеспечивающий связь с клиентом. При инициализации он создает по экземпляру классов SimStateClient и TMInterface, регистрирует интерфейс в клиенте, а также хранит обработанные данные о состоянии машины. С помощью своих методов он также позволяет обращаться к клиенту при необходимости ввода каких-либо данных или для их получения.

Для хранения обработанных данных о машине был создан класс SimState. Он имеет в себе поля, которые уже передаются в нейронную сеть. Основные хранимые данные:

* + linear\_speed – линейная скорость автомобиля, то есть скорость по основной оси машины.
  + angular\_speed – боковая скорость автомобиля, то есть скорость по оси машины, перпендикулярной основной.
  + distance\_to\_centerline – расстояние до центра дороги.
  + angle\_to\_centerline – угол основной оси машины относительно дороги, где 0 означает, что машина смотрит в сторону, в которую нужно двигаться, а 1 по модулю – что машина смотрим в обратную сторону.
  + next\_curve\_distance – расстояние до ближайшего поворота.
  + next\_curve\_direction – направление следующего поворота (-1 – влево, 1 – вправо)

Помимо сложных методов подсчета этих данных класс также нормализует их до отрезка от -1 до 1.

Во время разработки было создано несколько сущностей подобных классов, отвечающих за разные типы трасс. Самый простой для трека, состоящего из одного прямого участка прямо до финиша, принимал только первые 4 указанных поля, так как не было необходимости знать, сколько осталось до поворотов и их направление по причине их отсутствия.

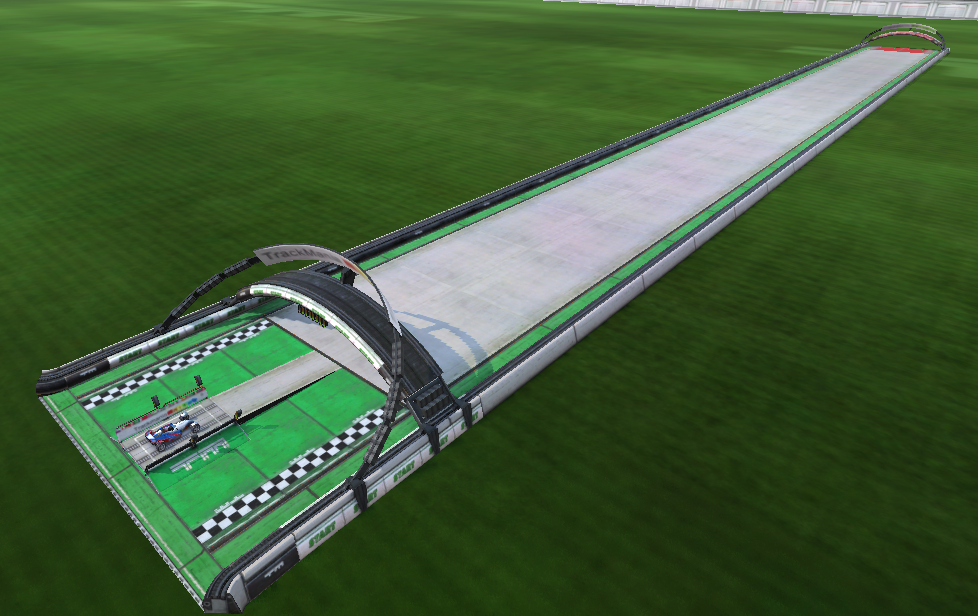


Рисунок 3.1. Прямая трасса

Класс, позволяющий получать данные для карты вида «змейка», имел в себе первые 6 указанных параметров, то есть теперь имел в себе значения, позволяющие понять, когда нужно поворачивать и в какую сторону.



Рисунок 3.2. Карта вида «змейка»

Для удобства разработки, средствами TMInterface были включены отображение текущего ввода и основные данные об автомобиле.



Рисунок 3.3. Интерфейс запущенного TMInterface

# 3.2. Создание gym окружения видеоигры для обучения

Для обучения нейронной сети было решено создать окружение с помощью библиотеки gymnasium. gymnasium – это библиотека для разработки и тестирования алгоритмов обучения с подкреплением. Она предоставляет различные окружения и возможности создания окружений, в которых агенты могут обучаться и взаимодействовать, что позволяет исследователям и разработчикам создавать, тестировать и сравнивать различные алгоритмы обучения с подкреплением.

Библиотека предоставляет класс Env, предполагаемый для наследования и переопределения методов обратного вызова. В связи с этим был создан класс TMEnv. Основные методы окружений gym:

* + step – метод, который вызывается каждый шаг алгоритма обучения и обычно вызывает остальные методы обратного вызова.
  + render – метод, обычно используемый для отображения каких-либо данных во время обучения.
  + reset – метод, предназначенный для приведения состояния окружения в свое обычное состояние.
  + reward – метод, возвращающий подкрепление, или награду, которое используется в алгоритме обучения.

Этот класс инициализирует и содержит в себе экземпляр класса SimStateInterface, что позволяет получать актуальные данные из состояния видеоигры и вводить действия для машины.

Самой сложной для создания являлась функция награды, так как от нее зависит то, как именно нейронная сеть будет менять веса. Было решено давать подкрепление за высокую скорость, высокое ускорение, прохождение трассы в правильном направлении, прохождение поворотов и трека в целом. Отрицательная награда даётся за отрицательную скорость, за непрохождение трассы или прохождение трассы и поворотов в неправильном направлении, за падение из трека и за факт слишком сильного поворота машины в неправильном направлении.

Другой важной функцией является функция step, вызываемая каждый шаг обучения. Она принимает ввод для видеоигры, получаемый от модели, обрабатывает его, вводит в игру, сохраняет полученные данные, проверяет, не нужно ли перезапустить обучение и перезапускает при необходимости, а также возвращает все необходимые для нейронной сети данные:

* + observation – обработанные и нормализованные данные, полученные из клиента (SimStateClient), интерфейса (SimStateInterface) и класса обработки состояния (SimState) видеоигры.
  + reward – награда, полученная алгоритмом обучения за текущий шаг.
  + done – булевая переменная, означающая, закончилась ли текущая сессия обучения и было ли перезапущено окружение.

Помимо этого, TMEnv также хранит множество вспомогательных параметров, большинство из которых позволяют функции награды более корректно определять подкрепление, как, например, previous\_distance\_to\_curve, отвечающая за расстояние до поворота в предыдущий шаг обучения и с которой сравнивается текущее расстояние до поворота.

Этот класс также обеспечивает факт запущенности видеоигры, устанавливает ее скорость, хранит в себе значения, используемые нейронной сетью. Основная из таких - observation\_space, определяющая, сколько входных и параметров будет принимать модель нейронной сети. Она имеет определение в виде

OBSERVATION\_SPACE = Box(-1, 1, (10,)).

Это означает, что нейросеть принимает 10 параметров, все из которых принимают значения в диапазоне от -1 до 1.

# 3.3. Реализация нейронной сети

Для обучения с подкреплением было решено использовать метод обучения DQN (Deep Q-Network). Его основными компонентами являются:

* + Q-значение – Q(s,a) представляет собой ожидаемую суммарную награду, которую агент получит, начиная из состояния s, выполняя действие a и следуя некоторой стратегии в будущем.
  + Нейронная сеть – для аппроксимации функций Q-значений используется глубокая нейронная сеть. Входом сети является состояние s, а на выходе сеть выдает Q-значения для всех возможных действий a в этом состоянии.
  + Целевая сеть – DQN использует две сети: основную (policy) сеть и целевую (target) сеть. Целевая сеть копирует веса основной сети через определенные интервалы времени. Это помогает стабилизировать обучение, так как целевые значения изменяются медленнее.
  + Память воспроизведения опыта (Replay Memory) – опыт агента (переходы) сохраняется в буфере воспроизведения. Переходы случайным образом выбираются из буфера для обучения. Это снижает корреляцию между последовательными шагами и делает обучение более стабильным и эффективным.

Далее каждый из компонентов будет описан более подробно.

В памяти воспроизведения опыта (Replay Memory) хранятся переходы, которые наблюдает агент, что позволяет повторно использовать эти данные в дальнейшем. Путем случайной выборки из нее переходы, составляющие партию, декоррелируются, что значительно стабилизирует и улучшает процедуру обучения сети.

Для этого понадобилось 2 класса:

* + Transition – именованный кортеж, представляющий один переход в окружении. Он сопоставляет пары (состояние, действие) с их результатом (следующее\_состояние, награда).
  + ReplayMemory – циклический буфер ограниченного размера, в котором хранятся переходы, наблюдавшиеся в последнее время. Он также реализует метод sample для выбора случайной партии переходов для обучения.

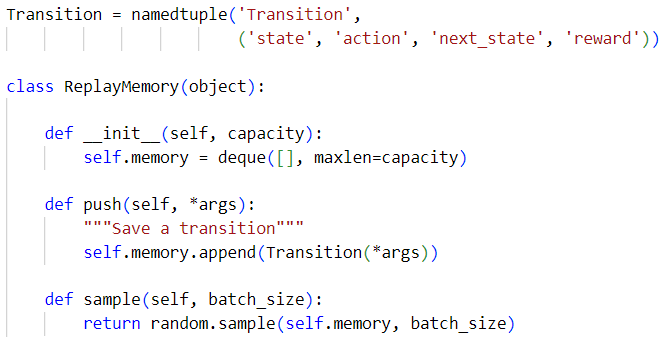


Рисунок 3.5. Реализация классов памяти воспроизведения опыта

Основной целью алгоритма обучения с подкреплением является обучение политики, которая пытается максимизировать дисконтированное кумулятивное вознаграждение

,

где дисконт γ является константой от 0 до 1. Низкие значения дисконта делают вознаграждения из неопределенного далекого будущего менее важными для агента, чем те, которые он может получить в ближайшем будущем и в которых он может быть достаточно уверен. Они также побуждают агентов получать вознаграждения ближе по времени, чем эквивалентные вознаграждения, которые находятся в будущем.

Основная идея Q-обучения заключается в том, что если бы была функция Q\*: State×Action→R, которая могла бы сказать, какой будет награда, если будет предпринято действие в данном состоянии, то можно было бы легко построить политику, которая максимизирует вознаграждение:

.

Однако агент не знает всего о мире, поэтому у него нет доступа к Q\*. Но поскольку нейронные сети являются универсальными аппроксиматорами функций, можно просто создать такую сеть и обучить ее, чтобы она напоминала Q\*.

Для правила обновления обучения было решено использовать факт, что каждая функция Q для некоторой политики подчиняется уравнению Беллмана:

.

Разность между двумя сторонами равенства называется погрешностью временной разности, δ:

.

Чтобы минимизировать эту ошибку, мы будем использовать потерю Хьюбера. Функция потерь Хьюбера действует как средняя квадратичная ошибка, когда ошибка мала, и как средняя абсолютная ошибка, когда ошибка велика – это делает его более устойчивым к выбросам, когда оценки Q очень зашумлены. Это вычисляется для пакетов переходов B, взятых из памяти воспроизведения опыта:

,

где

Из-за сложности представления всей физики видеоигры TrackMania Nations Forever было решено использовать Model-Free алгоритм обучения вместо Model-Based. Это означает, при обучении агент будет полагаться только на получаемые данные и не будет пытаться предсказать изменение состояния окружения. Таким образом, для того, чтобы понять, хорошим ли является действие, агенту сначала нужно будет его предпринять – он не сможет предсказать награду дальнейшего состояния видеоигры.

Модель представляет собой нейронную сеть с прямой передачей данных, которая учитывает разницу между текущим и предыдущим состояниями. У нее есть два выхода, представляющие собой Q(s,left) и Q(s,right) (где s - вход сети). По сути, сеть пытается предсказать ожидаемую прибыль от каждого действия с учетом текущих входных данных.

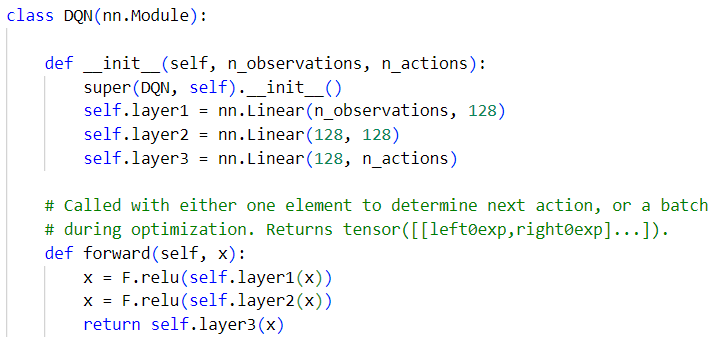


Рисунок 3.6. Реализация класса нейронной сети

Скрипт обучения имеет много различных параметров, основные из которых было решено выделить в отдельный класс, как гиперпараметры.

* + BATCH\_SIZE = 128 – размер пакета переходов, которые выбираются из памяти воспроизведения опыта для каждого шага обучения. Большой размер пакета может стабилизировать обучение за счет более точной оценки градиента, но также требует больше памяти и вычислительных ресурсов. Выбранный размер пакета в 128 является распространенным выбором, который балансирует эти факторы.
  + GAMMA = 0.99 – дисконтирующий фактор для подкрепления, описанный ранее.
  + EPS\_START = 0.95 – начальное значение эпсилон в эпсилон-жадной политике. Эпсилон управляет балансом между исследованием (пробованием новых действия) и эксплуатацией (выбором наилучшего известного действия). Высокое начальное значение эпсилон означает, что агент будет много исследовать в начале, пробуя разные действия для сбора информации об окружении.
  + EPS\_END = 0.05 – конечное значение эпсилон после его снижения. Это значение, до которого эпсилон уменьшится со временем. Низкое конечное значение эпсилон означает, что в конечном итоге агент будет в основном использовать то, что он уже выучил, но иногда будет исследовать, чтобы избежать застревания в субоптимальном поведении.
  + EPS\_DECAY = 2500 – скорость, с которой эпсилон уменьшается от EPS\_START до EPS\_END.

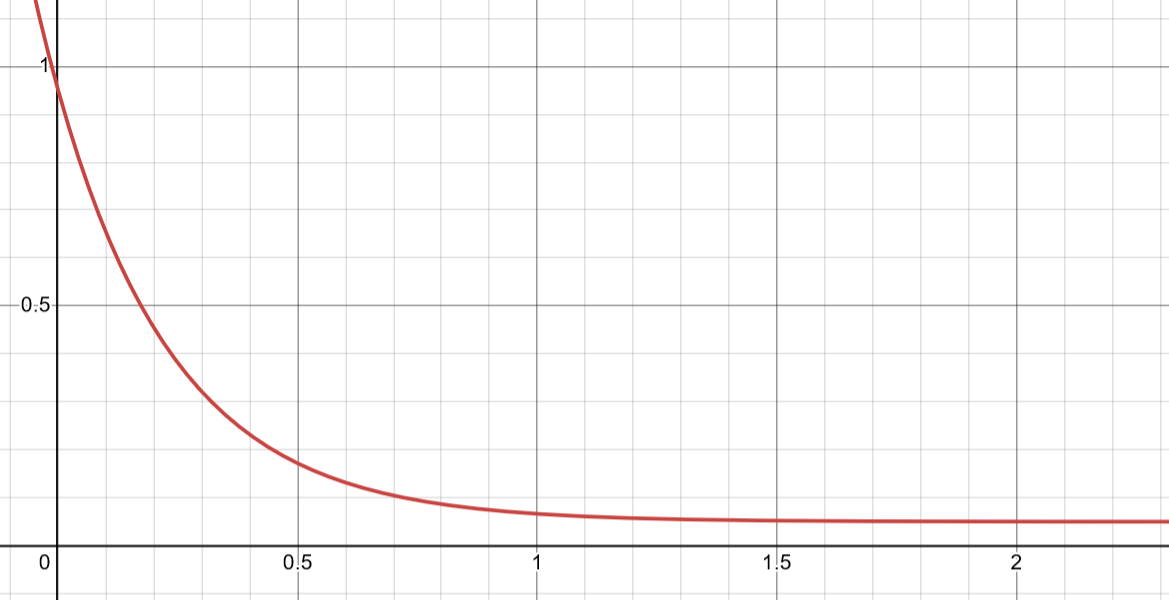


Рисунок 3.7. Функция эпсилон, где ось x – число выборов действий в десятках тысяч

* + TAU = 0.005 - скорость обновления целевой сети. Параметр τ управляет тем, как быстро целевая сеть обновляется, чтобы соответствовать основной сети. Малое значение τ означает, что целевая сеть обновляется медленно, что приводит к более стабильному обучению.
  + LR (Learning Rate) = 0.0001 - скорость обучения оптимизатора, конкретно AdamW. Скорость обучения определяет размер шагов, которые делает оптимизатор в направлении градиента. Меньшая скорость обучения означает, что агент будет делать более точные обновления параметров сети, что может привести к более стабильному и точному обучению, но это может также замедлить сходимость.

Основной функцией выбора действия является функция select\_action, которая выбирает действие в соответствии с эпсилон-жадной политикой.

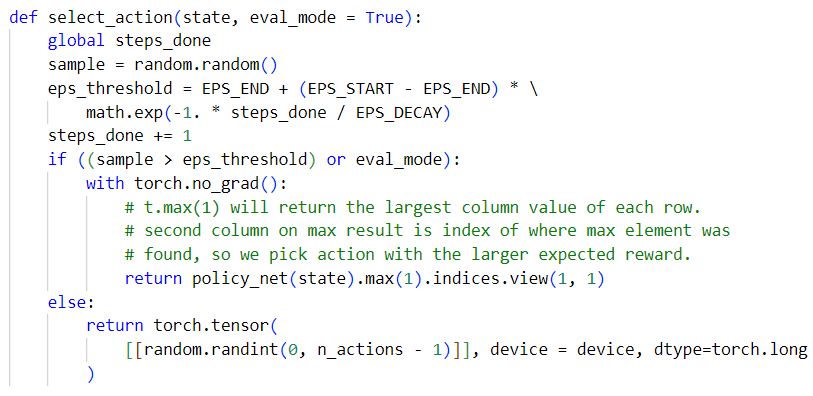


Рисунок 3.8. Реализация функции select\_action

Основное обучение нейронной сети происходит в функции optimize\_model, вызываемая каждый шаг обучения после совершения действия и получения результатов выполненного действия. Сначала она делает выборку, объединяет все тензоры в один, вычисляет Q(st,at) и V(st+1)=maxaQ(st+1,a) и объединяет их в нашу потерю. По определению, V(s)=0, если s - терминальное состояние. Для повышения устойчивости также используется целевая сеть для вычисления V(st+1). Целевая сеть обновляется на каждом шаге с помощью мягкого обновления, управляемого гиперпараметром TAU.

Обучение начинается с момента, когда в памяти переходов набирается больше элементов, чем прописано в гиперпараметре размера пакета переходов BATCH\_SIZE. Если это условие выполнено, то из памяти берутся случайные переходы и трансформируется в удобный формат.

После этого из выборки отдельно вычисляются состояния, которые являются конечными. В обучении с подкреплением конечное состояние — это состояние, в котором эпизод заканчивается. В нашем случае, конец эпизода — это финиш трассы или проигрыш, определяемый концом отведенного для прохождения временем или сильно отрицательной аккумулированной награды. В конечном состоянии нет следующего состояния, на которое можно было бы опираться для прогнозирования будущих наград. Поэтому в этих состояниях значение функции Q устанавливается равным 0.

После этого для всех состояний выборки с помощью основной модели создается массив значений Q – state\_action\_values. Он будет использован в функции потерь, как существующие значения функции.

Для неконечных состояний целевая сеть вычисляет значения V, которые потом используются для вычисления ожидаемых значений функции Q, как суммы награды и дисконтированного значения V для следующего состояния.

После этого функция потерь вычисляется с использованием потери Хьюбера. Затем оптимизатор обновляет параметры модели, используя вычисленный градиент. Градиенты ограничиваются по величине для предотвращения скачков значений.

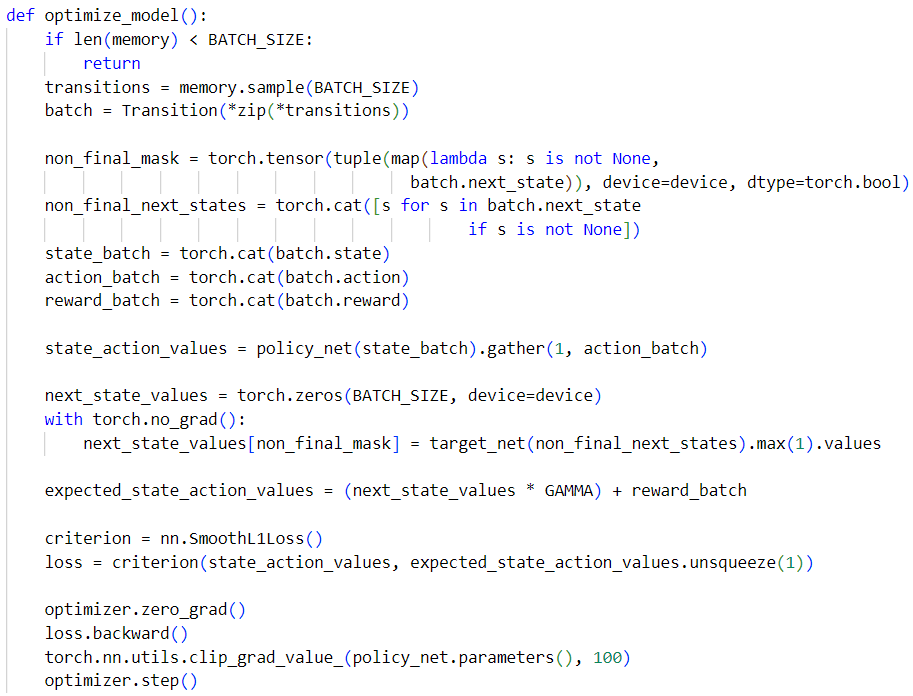


Рисунок 3.9. Реализация функции optimize\_model

Теперь можно определить основной цикл обучения. В начале окружение сбрасывается и получается тензор начального состояния. Затем выбирается и исполняется действие, наблюдается следующее состояние и вознаграждение, после чего модель оптимизируется. Когда эпизод заканчивается, мы цикл перезапускается.

Действия выбираются либо случайным образом, либо на с помощью модели, получая значения следующего шага из окружения. Мы записываем результаты в память воспроизведения, а также запускаем шаг оптимизации на каждой итерации. Оптимизация выбирает случайную выборку из памяти воспроизведения для обучения модели. «Старая» целевая сеть также используется в оптимизации для вычисления ожидаемых значений Q. Мягкое обновление ее весов выполняется на каждом шаге.

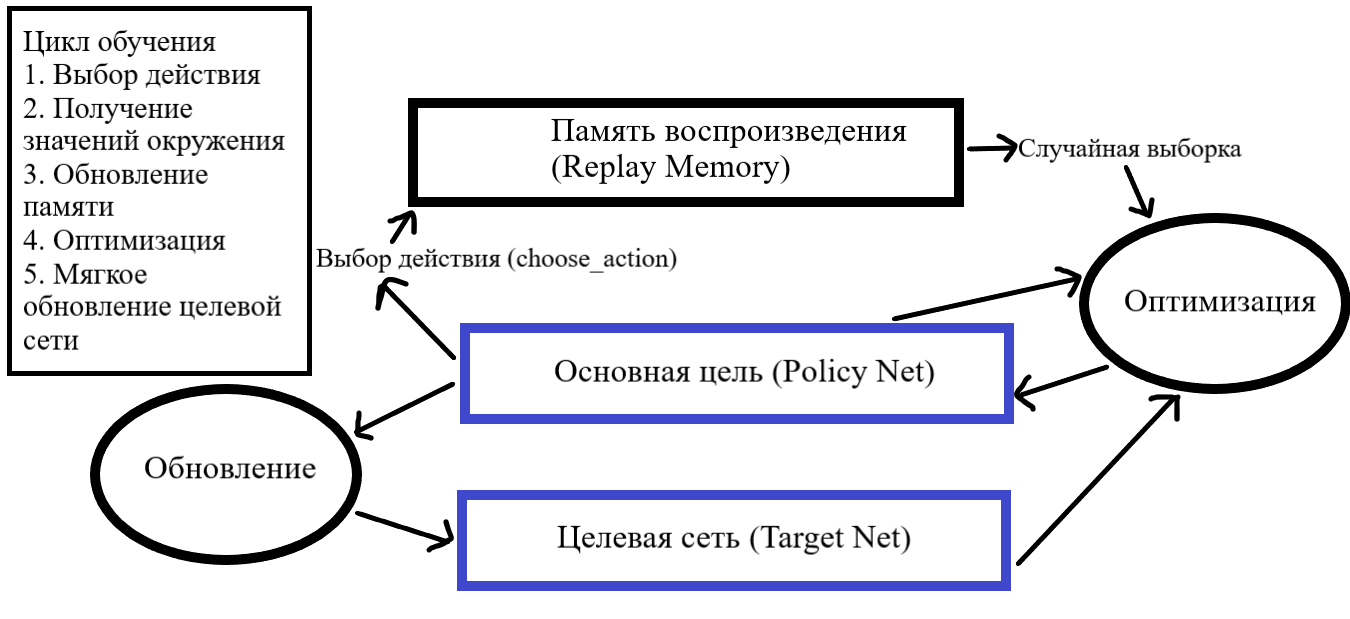


Рисунок 3.9. Упрощенная схема цикла обучения

Также были созданы средства для сохранения и загрузки моделей, чтобы в дальнейшем их использовать для последующего дообучения, переобучения или демонстрации.

Так, каждые 50 эпизодов сохраняются словари состояний обеих моделей и оптимизатора, вся память воспроизведения и количество выполненных шагов (для корректной работы эпсилон-жадной стратегии).

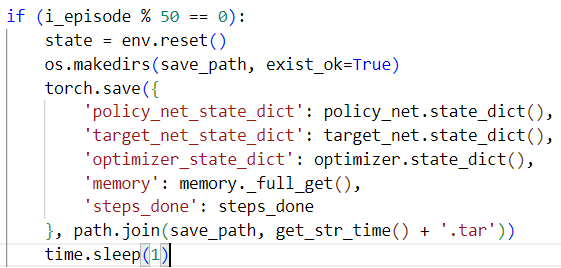


Рисунок 3.10. Реализация сохранения чекпоинта обучения

Для дообучения используются все данные, но для переобучения при необходимости можно указать свое число выполненных шагов, чтобы агент опять начал больше исследовать, чем эксплуатировать.

Для демонстрации достаточно лишь словаря состояний основной модели. Также при демонстрации убраны шаги оптимизации модели и использование полученной награды. Эпсилон-жадная стратегия тоже не имеет влияния, так как агент при демонстрации всегда эксплуатирует, а не исследует.

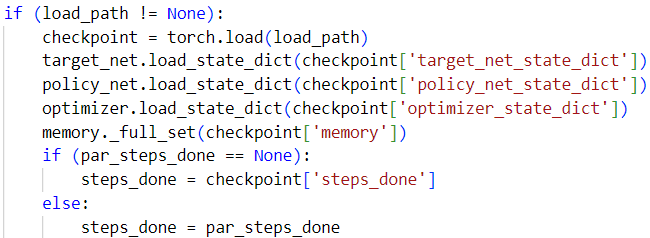


Рисунок 3.11. Реализация загрузки чекпоинта обучения

# 3.4. Заключение по приложению и циклу его работы

Цикл работы приложения в конечном итоге выглядит следующим образом:

* + Инициализируются основная и целевая модели, память воспроизведения и оптимизатор.
  + Запускается и инициализируется окружение, в котором будет происходить обучение.
  + Запускается и инициализируется интерфейс, связывающий клиент видеоигры и окружение.
  + Запускается и инициализируется клиент, связывающий интерфейс и видеоигру. Предполагается, что видеоигра запущена и выбран подходящий трек, на котором будет происходить обучение.
  + Начинается цикл обучения, в начале которого окружение сбрасывается и записывается тензор начального состояния.
  + Основываясь на состоянии, выбирается действие (либо случайное, либо выбранное моделью, что зависит от эпсилон), которое отправляется в окружение, после чего окружение отправляет данные о своем состоянии приложению.
  + Полученные от окружения данные о факте конца эпизода используются для конца эпизода.
  + Полученные от окружения данные и введенное управление добавляются в память воспроизведения, после чего используются для оптимизации модели целевой модели.
  + Обновленная целевая модель используется для мягкого обновления основной модели.
  + Полученные от окружения данные о состоянии видеоигры сохраняются для последующего выбора управления.

Получение состояния окружения происходит следующим образом:

* + Приложение делает запрос данных окружения путем сброса окружения или после ввода игрового управления.
  + При сбросе окружения оно обнуляет все свои значения, делает команду интерфейсу на перезапуск трека, после чего возвращает обнуленные данные приложению.
  + Иначе окружение принимает ввод и передает его интерфейсу.
  + Происходит ожидание в 0.05 секунд для изменения состояния видеоигры.
  + Окружение запрашивает данные о состоянии видеоигры у интерфейса.
  + Окружение вычисляет функцию награды на основе введенных данных.
  + Окружение проверяет, не должен ли эпизод обучения быть закончен (например, при финише трека, концу отведенного времени или отрицательной аккумулированной награде)
  + Окружение передает данные о состоянии видеоигры, награду и факт конца эпизода приложению.

Получение состояния видеоигры происходит следующим образом:

* + Окружение запрашивает данные о состоянии видеоигры у интерфейса.
  + Интерфейс запрашивает данные у клиента.
  + Клиент каждый физический тик игры сохраняет состояние видеоигры, благодаря чему может ответить на запрос интерфейса.
  + Интерфейс обрабатывает полученные данные, приводит их в нормализованное состояние и отвечает на запрос окружения.

Итоговая диаграмма классов без указания их полей и методов представлена на рисунке 3.12. Синим цветом были выделены классы библиотеки TMInterface, используемые в приложении; зеленым цветом классы, обеспечивающие связь классов библиотеки и обучающей части приложения; желтым цветом класс TMEnv, хранящий в себе gymnasium окружение; светло-зеленым цветом класс GameLauncher, запускающий видеоигру при начале обучения; синим цветом классы, хранящие в себе алгоритм обучения; розовым цветом классы, отвечающие за память воспроизведения опыта.

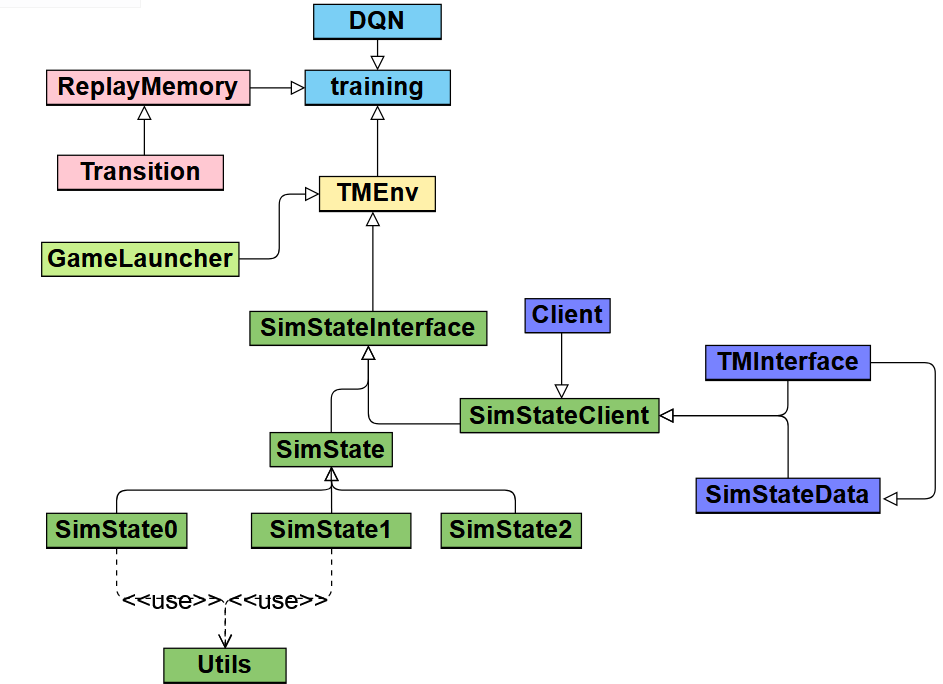


Рисунок 3.12. Диаграмма классов приложения

# 4. Обучение и тестирование моделей

# 4.1. Обучение на прямой карте

Для тестирования работы приложения было решено не начинать обучение на сложных картах с поворотами, а на прямом треке, на котором оптимальной стратегией является просто удерживание кнопки газа.

Изначально обучение происходило неудачно, так как модели были плохо настроены, а взаимодействие с игрой часто заканчивалось завершением работы приложения из-за отключения клиента и видеоигры. Недостаточное понимание темы и сути эпсилон-жадной стратегии также замедляли процесс.

Факт того, что при одновременно зажатых кнопках газа и тормоза машина продолжала ехать прямо, хоть и не так быстро, казался фактом эффективности обучения и того, что модель такими темпами научится проходить трек, когда на самом деле действия все еще были близки к случайным.

Когда проблемы были решены, обучение начало идти лучшим образом, благодаря чему были достигнуты позволительные для исследования результаты.

Так, на рисунке 4.1. видно, что при идеальной стратегии постоянного нажатия газа, то есть при идеальном времени в 7.82с, модель с каждым разом улучшала свой результат. На рисунке по оси y показано время в секундах, а по оси х – количество пройденных эпизодов, которое для масштаба было разделено на 100. Для получения этих результатов были загружены чекпоинты, сохраненные в разные моменты обучения, которые после использовались для демонстрации.

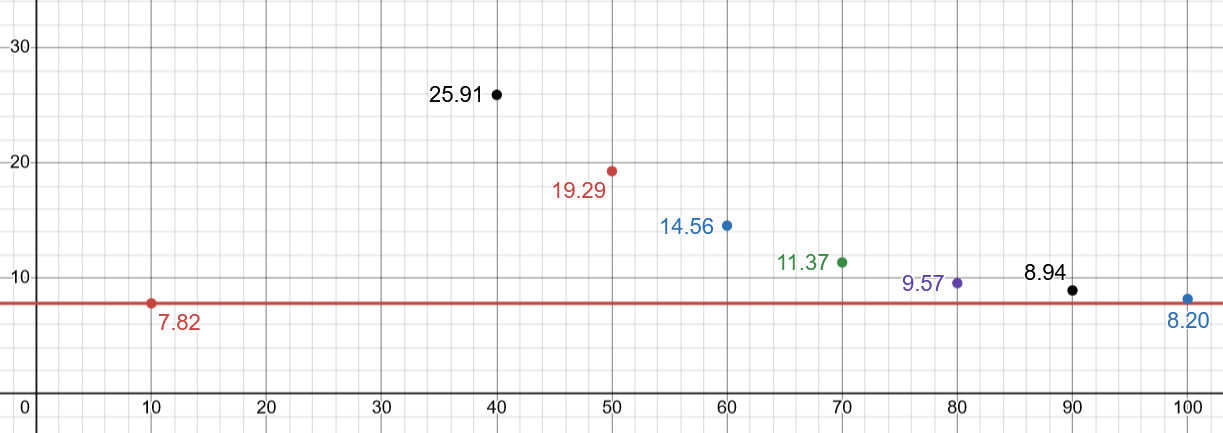


Рисунок 4.1. График обучения модели на прямой карте

4.2. Обучение на карте вида «змейка»

С получением успешных результатов на прямой карте, было решено перейти на обучение на карте вида «змейке». Она включала в себя множество повторяющихся поворотов, что предполагало повороты, а также использование тормоза для торможения перед ними.

Функция награды и окружение в целом также были изменена, чтобы учитывать данные, получаемые интерфейсом с такой карты.

С учетом прошлых ошибок и более сильного понимания темы и кода, обучение на этой карте прошло быстрее, что позволило машине быстро закончить трек. Проблемой было то, что машина часто врезалась в стены и проезжала трек, прижавшись к одной.

Из-за этого вопрос уже стоял не в том, чтобы создать приложение для создания моделей, которое позволит машине финишировать, а в том, чтобы это приложение создавало модели, которые создадут модели, способные финишировать быстро.



Рисунок 4.2. Прижатая к стенке машина

Для решения этой проблемы нужно было лучшее средство для тестирования окружения и функции награды. Был создан прострой скрипт, который также инициализирует окружение и имитирует процесс обучения для окружения. Единственным отличием было то, что управление совершалось игроком на клавиатуре, а данные от окружения использовались не для обучения, а для их визуализации.

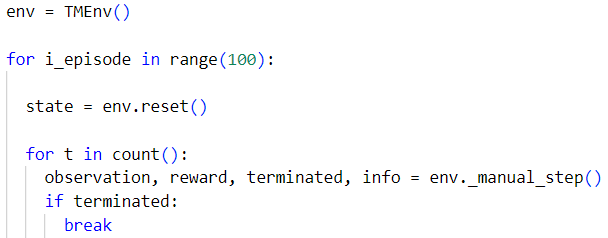


Рисунок 4.3. Скрипт имитации работы приложения

Это позволило явно видеть, какие именно награды получает приложение, что позволило изменять функцию подкрепления соответствующим образом, которое сильнее поощряло правильные действия и сильнее наказывало за ошибки.

Так, теперь тактика прижатия к стенке давала отрицательную награду, в то время как большая скорость сильнее вознаграждалась.

Благодаря этому были получены следующие результаты, представленные на рисунке 4.4. При моем результате, равном 14.47 секундам, лучший результат модели составлял 23.44 секунд, что соответствует техническому заданию.

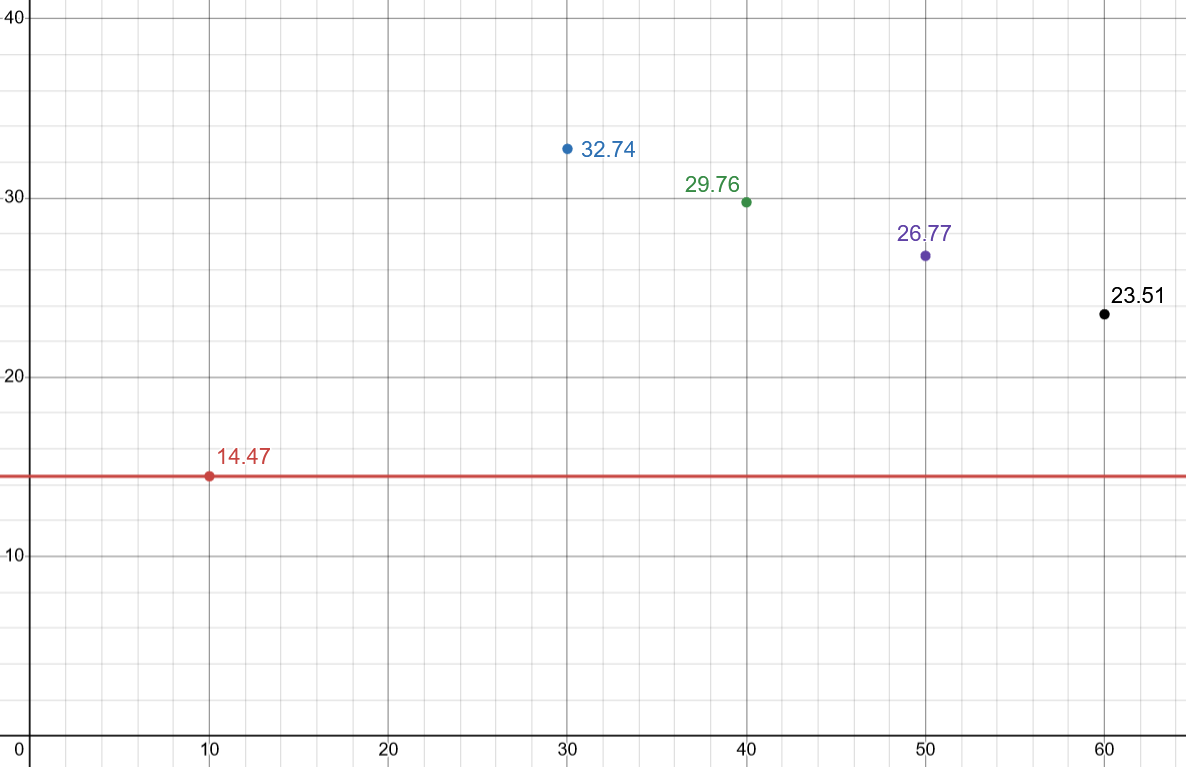


Рисунок 4.4. График одного из первых обучений на карте «змейка».

Тем не менее, модель не была лишена недостатков и имела потенциал для оптимизации, так как столкновения со стенами до сих пор случались.

Последующие попытки улучшить создаваемые модели не привели к улучшению времени прохождения трассы, но либо ускорили время, за которое модели создавались, либо дали информацию о лучшем выборе гиперпараметров и функции награды.

Например, были предприняты попытки обучения с очень награждающим подкреплением, и очень наказывающим. Первая предполагала почти полное отсутствие условий выставления отрицательной награды, а если она и выдавалась, то в ограниченном количестве. В результате, модели достигали примерно тех же результатов, что и первоначально.

С другой стороны, очень наказывающее подкрепление привело к тому, что модель предпочитала не двигаться или ехать очень медленно. В результате, после длительного обучения, модель не могла даже финишировать гонку, не говоря уже про то, чтобы сделать это быстро.

Так, было выявлено, что более поощряющая и менее наказывающая функция подкрепления является более правильным вариантом.

Также было проведено несколько тестов приложения с различным набором гиперпараметров.

Так, увеличение значения EPS\_DECAY давало приложению больше времени для обучения, что приводило к более стабильным результатам. Слишком малые значения быстро приводили эпсилон к минимальному значению, что не давало агенту исследовать достаточно долго, чтобы привыкнуть к трассе. Слишком большие же сильно затягивали обучение, что было неэффективным. В итоге, если оставлять обучение на 8 часов, то оптимальное значение EPS\_DECAY было выявлено в числе 50000.

Увеличение BATCH\_SIZE также дало положительный результат. Слишком малые значения не давали достаточно данных для эффективной оптимизации моделей, а слишком большие создавали проблемы с производительностью. Было решено установить значение BATCH\_SIZE в 192, что было оптимальным как в плане эффективности оптимизации, так и производительности.

GAMMA осталась на том же уровне в 0.9. Значения, еще более близкие к 1 или, наоборот, меньшие часто приводили к тому, что машина врезалась в стены. Меньшие значения поощряли это тем, что быстрая скорость давала моментальную награду сейчас, а возможная будущая награда игнорировалась. В связи с этим торможения или отпускания газа на поворотах происходило реже и реже пропорционально GAMMA. Высокие значения же поощряли найденные рабочие тактики, которые известно приводили к большим наградам, даже если они не были оптимальны. Значение в 0.9 позволило соблюсти баланс между этими двумя проблемами.

Также были предприняты попытки изменения самой структуры нейронной сети. Изменялись количество слоев, их размер и функции между ними. Во всех случаях результаты выходили хуже, чем при оригинальной структуре. Слишком большие размеры сети приводили к переобучению, что сильно понижало адаптируемость модели, что в свою очередь заметно замедляло обучение. Малая сеть же просто не предоставляла достаточной гибкости для такого окружения. Модели обучались до своей лучшей стратегии довольно быстро, но эти стратегии часто были далеки от оптимальных.

Таким образом, после всех изменений параметров, влияющих на обучение, были достигнуты результаты, представленные на рисунке 4.5. Модель так и не смогла обучиться достаточно, чтобы суметь обогнать человека, ранее игравшего в гоночные игры, но достаточно, чтобы суметь финишировать трассу, используя субоптимальные стратегии. Машина все еще иногда врезалась в стены, но намного чаще использовала тормоз и отпускание газа. Был достигнут лучший результат в 18.86 секунд, который был сочтен удовлетворительным для исследования.

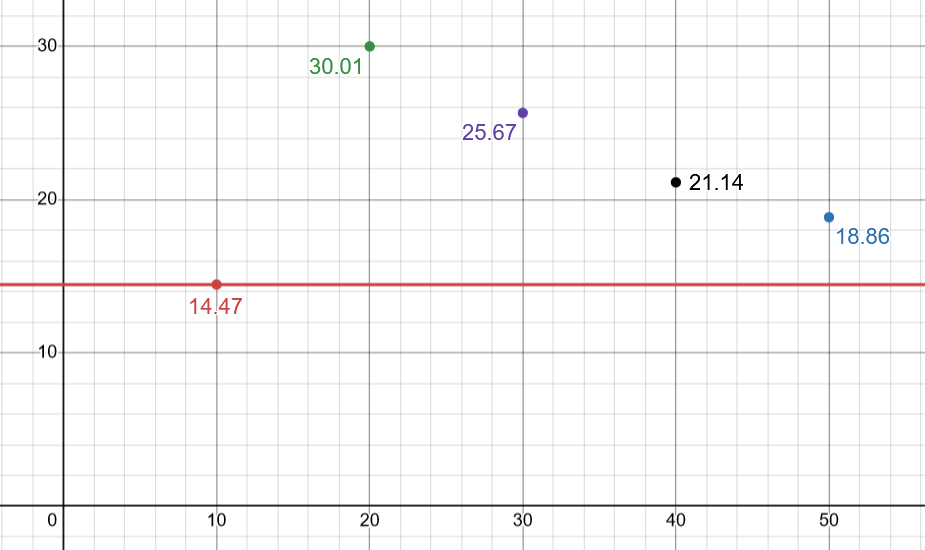


Рисунок 4.5. График обучения лучшей модели на карте «змейка»

Неидеальность создаваемых моделей говорит о том, что приложение имеет потенциал для дальнейшего развития. Дальнейшие опыты с различными наборами гиперпараметров, архитектуры нейронной сети и функции подкрепления могут привести к еще более значительным результатам, в которые модели могут быстро финишировать уже представленные в работе трассы, так и успешно проходить более комплексные со сложными поворотами.

# 5. Заключение

В результате выполнения ВКР была изучена и проанализирована предметная область, изучены информационные ресурсы схожей тематики, выявлены недостатки и определены преимущества существующих аналогов. В качестве аналогов были выбраны существующие решения в сериях видеоигр Need For Speed, Forza и TrackMania, созданные как разработчиками, так и обычными пользователями. В процессе анализа предметной области было сформировано техническое задание, включающее функциональные требования и требования к моделям.

Основными параметрами приложения является скорость обучения моделей, которое не должно превышать 8 часов, и устойчивость ко всем возможным событиям и ошибкам во время обучения. Была продумана архитектура взаимодействия между приложением и видеоигрой, а также между различными компонентами приложения. Были спроектированы функции и классы, отвечающие за обучение моделей и их работу с видеоигрой. Были рассмотрены различны средства разработки и выбраны наиболее подходящие из них. В процессе анализа была составлена ER-диаграмма, а также схема работы приложения.

В ходе разработки приложения был успешно применен и реализован метод обучения с подкреплением DQN, который позволил создать эффективное приложение для создания сложных моделей. В ходе его реализации были созданы такие вспомогательные средства, как память воспроизведения опыта и функции для загрузки и сохранения моделей, которые позволили сделать обучение еще более эффективным.

Была успешно использована библиотека gymnasium, которая позволила создать эффективное окружение для обучения моделей. Были реализованы все основные функции gymnasium-окружения, такие как observe, step и reward. Связь приложения и видеоигры была реализована с помощью интерфейса TMInterface, который позволял работать с данными игры напрямую, давая возможность свободно получать их или менять.

В ходе эксплуатации приложения производились многочисленные улучшения выбора гиперпараметров и функции подкрепления. Благодаря ним удалось добиться значительных результатов, среди которых скорость создания и обучения моделей, а также их эффективность и адаптируемость на создаваемых трассах.

Разработанное приложение и модели были проверены с помощью ручных методов тестирования. Основными метриками послужили скорость создания моделей, адаптируемость модели к новым и незнакомым ситуациям и скорость прохождения моделями трасс. По итогам тестирования и соответствующего корректирования работы приложения эффективность моделей и обучения возросла, а полученный результат посчитался удовлетворительным.

В результате выполнения ВКР были усовершенствованы навыки работы с языком программирования Python, библиотекой для создания моделей нейронной сети PyTorch, библиотекой для создания окружения для обучения с подкреплением gymnasium и интерфейсом для видеоигры TMInterface.

Реализованное приложение имеет большой потенциал для дальнейшего развития и конкуренции с существующими решениями. Есть простор для увеличения эффективности алгоритмов обучения, чтобы модели могли быстрее проезжать создаваемые трассы, а также могли заканчивать более комплексные треки.

# Список использованных источников

1. Exploration Strategies in Deep Reinforcement Learning [Электронный ресурс]. URL: https://lilianweng.github.io/posts/2020-06-07-exploration-drl/ (дата обращения: 10.04.2024).
2. Epsilon-Greedy Algorithm in Reinforcement Learning [Электронный ресурс]. URL: https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/ (дата обращения: 10.04.2024)
3. Reinforcement Learning (DQN) Tutorial [Электронный ресурс]. URL: https://pytorch.org/tutorials/intermediate/reinforcement\_q\_learning.html (дата обращения: 11.04.2024)
4. TMInterface [Электронный ресурс]. URL: https://donadigo.com/tminterface/ (дата обращения: 14.04.2024)
5. Save and Load the Model [Электронный ресурс]. URL: https://pytorch.org/tutorials/beginner/basics/saveloadrun\_tutorial.html (дата обращения: 17.04.2024)
6. Saving and loading a general checkpoint in PyTorch   
   [Электронный ресурс]. URL: https://pytorch.org/tutorials/recipes/recipes/saving\_and\_loading\_a\_general\_checkpoint.html (дата обращения: 17.04.2024)
7. Tips for Loading an nn.Module from a Checkpoint   
   [Электронный ресурс]. URL: https://pytorch.org/tutorials/recipes/recipes/module\_load\_state\_dict\_tips.html?highlight=loading%20nn%20module%20from%20checkpoint (дата обращения: 17.04.2024)
8. Saving and Loading Models [Электронный ресурс]. URL: https://pytorch.org/tutorials/beginner/saving\_loading\_models.html (дата обращения: 17.04.2024)
9. How Forza's Drivatar Actually Works | AI and Games #60 [Электронный ресурс]. URL: https://www.youtube.com/watch?v=JeYP9eyIl4E&t=1s (дата обращения: 30.04.2024)
10. Drivatar in Forza Motorsport [Электронный ресурс]. URL: http://web.archive.org/web/20140208083127/http://research.microsoft.com/en-us/projects/drivatar/forza.aspx (дата обращения: 30.04.2024)
11. TMRL [Электронный ресурс]. URL: https://github.com/trackmania-rl/tmrl (дата обращения: 10.05.2024)
12. Функция потерь Хьюбера [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F\_%D0%BF%D0%BE%D1%82%D0%B5%D1%80%D1%8C\_%D0%A5%D1%8C%D1%8E%D0%B1%D0%B5%D1%80%D0%B0 (дата обращения: 15.05.2024)
13. Model-Based and Model-Free Reinforcement Learning: Pytennis Case Study [Электронный ресурс]. URL: https://neptune.ai/blog/model-based-and-model-free-reinforcement-learning-pytennis-case-study (дата обращения: 18.05.2024)